

[Sign in](#)

egorov-m / algForExam Public

Алгоритмы: Материалы для экзамена.

MIT license

2 stars 2 forks Branches Tags Activity

Star

Notifications

Code Issues Pull requests Actions Projects Security

main



Go to file

Code

egorov-m	Added CONTRIBUTING.md	efb011e · last year
algForExam	Added Dijkstra code	2 years ago
.gitignore	Initial commit	2 years ago
CONTRIBUTING.md	Added CONTRIBUTING.md	last year
LICENSE	Initial commit	2 years ago
README.md	Added CONTRIBUTING.md	last year
algForExam.sln	Initial VS Console project	2 years ago

README MIT license

Алгоритмы: Материалы для экзамена.

С инструкцией о том, как вносить вклад в репозиторий, можно ознакомиться в [руководстве](#).

Оглавление

- [Оглавление](#)

- [Билет 1: Пузырьковая сортировка \(Bubble Sort\)](#)
 - [Описание](#)
 - [Реализация \(C# пример\)](#)
- [Билет 2: Сортировка вставками \(Insertion Sort\)](#)
 - [Описание](#)
 - [Реализация \(C# пример\)](#)
- [Билет 3: Сортировка выбором \(Selection Sort\)](#)
 - [Описание](#)
 - [Реализация \(C# пример\)](#)
- [Билет 4: "Шейкерная" сортировка \(Сортировка перемешиванием, Двухнаправленная сортировка, Cocktail Sort\)](#)
 - [Описание](#)
 - [Реализация \(C# пример\)](#)
- [Билет 5: Сортировка Шелла \(Shell Sort\)](#)
 - [Описание](#)
 - [Реализация \(C# пример\)](#)
- [Билет 6: Алгоритм бинарного \(двоичного\) поиска \(Binary Search\)](#)
 - [Описание](#)
 - [Реализация \(C# пример\)](#)
- [Билет 7: Быстрая сортировка \(Quick Sort\)](#)
 - [Описание](#)
 - [Реализация \(C# пример\)](#)
- [Билет 8: Внешняя сортировка слиянием \(External merge sort\)](#)
 - [Описание](#)
 - [Реализация \(C# пример\)](#)
- [Билет 9: Сортировка с помощью двоичного дерева \(Tree Sort\)](#)
 - [Описание](#)
 - [Реализация \(C# пример\)](#)
- [Билет 10: Поразрядная сортировка \(Radix Sort\)](#)
 - [Описание](#)
 - [Реализация \(C# пример\)](#)
- [Билет 11: Хэш-таблицы с разрешением коллизий методом цепочек](#)
 - [Описание](#)
 - [Реализация \(C# пример\)](#)
- [Билет 12: Хэш-таблицы с разрешением коллизий методом открытой адресации](#)
 - [Описание](#)
 - [Реализация \(C# пример\)](#)
- [Билет 13: ABC сортировка для строк \(Allen Beechick Character Sort for string\)](#)

- [Описание](#)
- [Реализация \(C# пример\)](#)
- [Билет 14: Реализовать стек и базовые операции работы со стеком, с использованием собственного двусвязного списка](#)
 - [Описание](#)
 - [Реализация \(C# пример\)](#)
- [Билет 15: Реализовать очередь и базовые операции работы с очередью, с использованием собственного двусвязного списка](#)
 - [Описание](#)
 - [Реализация \(C# пример\)](#)
- [Билет 16: Алгоритм Крускала \(поиска минимального остовного дерева\)](#)
 - [Описание](#)
 - [Реализация \(C# пример\)](#)
- [Билет 17: Алгоритм Прима \(поиска минимального остовного дерева\)](#)
 - [Описание](#)
 - [Реализация \(C# пример\)](#)
- [Билет 18: Обход графа в глубину \(Depth-First Search, DFS\)](#)
 - [Описание](#)
 - [Реализация \(C# пример\)](#)
- [Билет 19: Обход графа в ширину \(Breadth-first search, BFS\)](#)
 - [Описание](#)
 - [Реализация \(C# пример\)](#)
- [Билет 20: Алгоритм Дейкстры \(поиск кратчайшего пути\) \(Dijkstra's algorithm\)](#)
 - [Описание](#)
 - [Реализация \(C# пример\)](#)

Билет 1: Пузырьковая сортировка (Bubble Sort)

Описание

Простейший алгоритм сортировки, который многократно меняет местами соседние элементы, если они расположены в неправильном порядке. Проход по списку повторяется до тех пор, пока список не будет отсортирован.

Сложность: $O(n^2)$, в лучшем случае (коллекция уже отсортирована) $O(n)$, алгоритм не подходит для большинства наборов данных, поскольку его временная сложность в среднем и худшем случае довольно высока.

Вспомогательное пространство: $O(1)$.

Дополнительно: <https://www.geeksforgeeks.org/bubble-sort/>

Реализация (C# пример)

```
public static IList<T> BubbleSort<T>(this IList<T> collection) where T : IC
{
    for (var i = 0; i < collection.Count; i++)
    {
        for (var j = i + 1; j < collection.Count; j++)
        {
            if (collection[i].CompareTo(collection[j]) > 0) // Сортировка п
            {
                (collection[j], collection[i]) = (collection[i], collection
            }
        }
    }

    return collection;
}
```

Билет 2: Сортировка вставками (Insertion Sort)

Описание

Простой алгоритм сортировки, который работает аналогично тому, как происходит сортировка игровых карт в руках. Массив мысленно разделён на отсортированную и неотсортированную части. Значения из неотсортированной части выбираются и помещаются в правильную позицию в отсортированной части.

Этот алгоритм является одним из самых простых в реализации, эффективен для небольших наборов данных, носит адаптивный характер: подходит для частично отсортированных наборов.

Сложность:

- Лучший случай: $O(n)$, возникает в случае, если массив уже отсортирован;
- Средний случай: $O(n^2)$, элементы перемешаны в порядке так, что порядок не является должным образом возрастающим или убывающим;
- Худший случай: $O(n^2)$, возникает в случае, когда массив уже отсортирован в обратном порядке;

Вспомогательное пространство: $O(1)$.

Дополнительно: <https://www.geeksforgeeks.org/insertion-sort/>

Реализация (C# пример)

```
public static IList<T> InsertionSort<T>(this IList<T> collection) where T :   
{  
    for (var i = 1; i < collection.Count; i++)  
    {  
        var item = collection[i];  
        var j = i - 1;  
  
        while (j >= 0 && collection[j].CompareTo(item) > 0) // Сортировка по  
        {  
            collection[j + 1] = collection[j];  
            j--;  
        }  
  
        collection[j + 1] = item;  
    }  
  
    return collection;  
}
```

Билет 3: Сортировка выбором (Selection Sort)

Описание

Алгоритм сортирует массив, многократно находя минимальный элемент (с учётом возрастания) из несортированной части и помещая его в начало. Алгоритм поддерживает два под массива в заданном массиве: под массив, который уже отсортирован, оставшийся под массив — неотсортированный.

По-умолчанию реализация алгоритма не [стабильна*](#). Однако этого можно достичь путём замены операции смены элементов местами на операцию толкания элементов вперёд.

Стабильный алгоритм:

Тот алгоритм, который не меняет порядок элементов с одинаковыми ключами относительно друг друга.

Сложность: $O(n^2)$ — во всех случаях, так-как есть два вложенных цикла: один цикл для выбора элемента массива один за другим $O(n)$, другой цикл для сравнения этого элемента с любым другим $O(n)$.

Вспомогательное пространство: $O(1)$.

Дополнительно: <https://www.geeksforgeeks.org/selection-sort/>,
<https://www.geeksforgeeks.org/stable-selection-sort/>

Реализация (C# пример)

```
public static IList<T> SelectionSort<T>(this IList<T> collection) where T :   
{  
    for (var i = 0; i < collection.Count - 1; i++)  
    {  
        var index = i;  
        for (var j = i + 1; j < collection.Count; j++)  
        {  
            if (collection[index].CompareTo(collection[j]) > 0) // Сортиров  
            {  
                index = j;  
            }  
        }  
  
        (collection[index], collection[i]) = (collection[i], collection[ind  
    }  
  
    return collection;  
}
```

Билет 4: “Шейкерная” сортировка (Сортировка перемешиванием, Двухнаправленная сортировка, Cocktail Sort)

Описание

Разновидность [пузырьковой сортировки](#). Алгоритм пузырьковой сортировки всегда обходит элементы слева и перемещает самый большой элемент в правильное положение на первой итерации, второй по величине — на второй и так далее. Коктейльная сортировка попеременно проходит через заданный массив в обоих направлениях. Коктейльная сортировка не требует ненужных итераций, что делает ее эффективной для больших массивов.

Каждая итерация алгоритма разбивается на два этапа:

1. Первый этап перебирает массив слева направо, как и при пузырьковой сортировке. Во время цикла сравниваются соседние элементы, и если значение слева больше значения справа, значения меняются местами. В конце

первой итерации наибольшее число будет находиться в конце массива.

2. Второй этап проходит по массиву в обратном направлении — начиная с элемента, непосредственно предшествующего последнему отсортированному элементу, и возвращаясь к началу массива. Здесь также сравниваются соседние элементы и при необходимости меняются местами.

Сравнение с пузырьковой сортировкой: временная сложность такая же, но Коктейльная сортировка работает лучше, чем пузырьковая, обычно мене чем в два раза быстрее.

Сложность: $O(n^2)$, в лучшем случае (коллекция уже отсортирована) $O(n)$.

Вспомогательное пространство: $O(1)$.

Дополнительно: <https://www.geeksforgeeks.org/cocktail-sort/>

Реализация (C# пример)

```
public static IList<T> CocktailSort<T>(this IList<T> collection) where T :
{
    for (var i = 0; i < collection.Count; i++)
    {
        var needSort = false;
        for (var j = i; j < collection.Count - 1 - i; j++)
        {
            if (collection[j].CompareTo(collection[j + 1]) > 0) // Сортиров
            {
                (collection[j], collection[j + 1]) = (collection[j + 1], co
                needSort = true;
            }
        }

        for (var j = collection.Count - 2 - i; j > i; j--)
        {
            if (collection[j - 1].CompareTo(collection[j]) > 0) // Сортиров
            {
                (collection[j], collection[j - 1]) = (collection[j - 1], co
                needSort = true;
            }
        }

        if (!needSort) return collection;
    }

    return collection;
}
```

Билет 5: Сортировка Шелла (Shell Sort)

Описание

Это обобщённая версия [алгоритма сортировки вставками](#). Сначала сортируются элементы, находящиеся далеко друг от друга, и последовательно уменьшается интервал между сортируемыми элементами, таким образом выполняется меньше обменов.

Интервал между сортируемыми элементами сокращается в зависимости от используемой последовательности. Вот несколько оптимальных последовательностей:

- Оригинальная последовательность Shell: $N/2, N/4, \dots, 1$;
- Последовательность Кнута: $1, 4, 13, \dots, (3^k - 1) / 2$;
- Последовательность Седжвика: $1, 8, 23, 77, 281, 1073, 4193, 16577 \dots 4j+1 + 3 \cdot 2^j + 1$;
- Приращение Хиббарда: $1, 3, 7, 15, 31, 63, 127, 255 \dots 511, \dots$;
- Последовательность Папернова и Стасевича: $1, 3, 5, 9, 17, 33, 65, \dots$;
- Последовательность Пратт: $1, 2, 3, 4, 6, 9, 8, 12, 18, 27, 16, 24, 36, 54, 81, \dots$;

Сложность:

- Лучший случай: $O(n \cdot \log(n))$, возникает в случае, если массив уже отсортирован, общее количество сравнений для каждого интервала равно размеру массива;
- Средний случай: $O(n \cdot \log(n))$, около $O(n^{1.25})$;
- Худший случай: $O(n^2)$, согласно теореме Пунена — сложность равна что-то вроде: $O(n \cdot \log(n))^2 / (\log(\log(n))^2)$ или $O(n \cdot \log(n))^2 / \log(\log(n))$ или $O(n \cdot (\log(n))^2)$

Сложность зависит от выбранной последовательности, для каждой конкретной отличается. Лучшая последовательность неизвестна.

Вспомогательное пространство: $O(1)$.

Дополнительно: <https://www.geeksforgeeks.org/shellsort>,
<https://www.programiz.com/dsa/shell-sort>

Реализация (C# пример)

```
public static IList<T> ShellSort<T>(this IList<T> collection) where T : IComparable<T>
{
    var step = collection.Count / 2;
```



```

while (step > 0)
{
    step /= 2;
    for (var i = step; i < collection.Count; i++)
    {
        var j = i;
        while (j >= step && collection[j - step].CompareTo(collection[j]) > 0)
        {
            (collection[j], collection[j - step]) = (collection[j - step], collection[j]);
            j -= step;
        }
    }
}

return collection;
}

```

Билет 6: Алгоритм бинарного (двоичного) поиска (Binary Search)

Описание

Алгоритм поиска, используемый в отсортированном массиве путём многократного деления интервала поиска пополам. Идея состоит в том, чтобы использовать информацию о том, что массив отсортирован, и уменьшить временную сложность до $O(\log(n))$.

Может быть реализован двумя способами: итерационный метод, рекурсивный метод — подход "разделяй и властвуй".

Сложность (рекурсивный, итеративный метод): $O(\log(n))$, в лучшем случае $O(1)$.

Вспомогательное пространство (рекурсивный метод): $O(\log(n))$.

Вспомогательное пространство (итеративный метод): $O(1)$.

Дополнительно: <https://www.geeksforgeeks.org/binary-search/>,
<https://www.programiz.com/dsa/binary-search>

Реализация (C# пример)

Рекурсивный метод

```

private static int BinarySearchRecursive<T>(this IList<T> collection, T key
{

```

```

        if (right < left) throw new ArgumentOutOfRangeException("Левая / правая граница указана неверно");

        var mid = left + (right - left) / 2; // ! вычислять нужно именно так
        var value = collection[mid];

        if (key.Equals(value)) return mid;

        return value.CompareTo(key) > 0 // Коллекция отсортирована по возрастанию
            ? collection.BinarySearchRecursive(key, left, mid - 1)
            : collection.BinarySearchRecursive(key, mid + 1, right);
    }

```

Итеративный метод

```

private static int BinarySearchIterative<T>(this IList<T> collection, T key)
{
    while (left <= right)
    {
        var mid = left + (right - left) / 2; // ! вычислять нужно именно так
        var value = collection[mid];

        if (key.Equals(value)) return mid;

        if (value.CompareTo(key) > 0) // Коллекция отсортирована по возрастанию
        {
            right = mid - 1;
        }
        else
        {
            left = mid + 1;
        }
    }

    throw new ArgumentOutOfRangeException("Левая / правая граница указана неверно");
}

```

Билет 7: Быстрая сортировка (Quick Sort)

Описание

Алгоритм быстрой сортировки представляет собой алгоритм «разделяй и властвуй». Первоначально он выбирает элемент в качестве опорного элемента и разбивает данный массив вокруг выбранного опорного элемента. Существует много разных версий QuickSort, которые по-разному выбирают точку опоры: 1. всегда выбирайте первый элемент в качестве опорного, 2. всегда выбирайте

последний элемент в качестве опорного, 3. выберите случайный элемент в качестве точки опоры, 4. выберите медиану в качестве точки опоры. Также есть несколько схем разделения.

Схема разделения Ломута

Предполагается, что опорный элемент является последним. Теперь инициализируется два счётчика: i и j . Выполняется итерация по массиву, увеличивается i , если $array[i] \leq pivot$ ($array[i] > pivot$), и заменяется $array[i]$ на $array[j]$, в противном случае увеличивается только счётчик j . После выхода из цикла, меняются местами $array[i]$ и $array[pivotIndex]$.

Схема разделения Хоара

(в целом более эффективен - в среднем делает в три раза меньше свопов):
Работает по принципу инициализации двух указателей, которые указывают на массив с начала и конца. Они двигаются друг к другу до тех пор, пока не будет найдена ситуация, когда меньше(больше) значение справа, а больше(меньше) значение слева, относительно опорного элемента. После этого два значения меняются местами.

Замечание: Если в качестве опорного элемента выбирать последний, то схема разделения Хоара может привести к тому, что QuickSort уйдёт в бесконечную рекурсию, в этом случае нужно будет произвести замену элемента.

Сложность:

- Лучший случай: $O(n * \log(n))$, возникает в случае, когда опорный элемент является средним элементом или рядом со средним;
- Худший случай: $O(n^2)$, возникает в случае, когда выбранный опорный элемент является самым большим или самым маленьким;
- Средний случай: $O(n * \log(n))$, происходит в случае, когда вышеуказанные условия не возникают;

Дополнительно: <https://www.geeksforgeeks.org/quick-sort/>,
<https://www.geeksforgeeks.org/hoares-vs-lomuto-partition-scheme-quicksort/>,
<https://www.programiz.com/dsa/quick-sort>

Реализация (C# пример)

Метод разделения Ломута

```
private static IList<T> QuickSortLomuto<T>(this IList<T> collection, int le   
{
```

```

        if (left >= right) return collection;
        var pivot = collection.PartitionLomuto(left, right);
        QuickSortLomuto(collection, left, pivot - 1);
        QuickSortLomuto(collection, pivot + 1, right);

        return collection;
    }

    private static int PartitionLomuto<T>(this IList<T> collection, int left, int right)
    {
        var pivot = left - 1; // В качестве опорного элемента выбирается самый левый элемент.
        for (var i = left; i < right; i++)
        {
            if (collection[right].CompareTo(collection[i]) > 0) // Сортировка по возрастанию
            {
                pivot++;
                (collection[pivot], collection[i]) = (collection[i], collection[pivot]);
            }
        }

        pivot++;
        (collection[pivot], collection[right]) = (collection[right], collection[pivot]);

        return pivot;
    }

```

Метод разделения Хоара

```

    private static IList<T> QuickSortHoare<T>(this IList<T> collection, int left, int right)
    {
        if (left < right)
        {
            var pivot = collection.PartitionHoare(left, right);
            collection.QuickSortHoare(left, pivot);
            collection.QuickSortHoare(pivot + 1, right);
        }

        return collection;
    }

    private static int PartitionHoare<T>(this IList<T> collection, int left, int right)
    {
        var pivot = collection[left]; // В качестве опорного элемента выбирается первый элемент.

        var i = left - 1;
        var j = right + 1;

        while (true)
        {

```

```
do
{
    i++;
} while (pivot.CompareTo(collection[i]) > 0); // Сортировка по возр

do
{
    j--;
} while (collection[j].CompareTo(pivot) > 0); // Сортировка по возр

if (i >= j)
{
    return j;
}

(collection[i], collection[j]) = (collection[j], collection[i]);
}
}
```

Билет 8: Внешняя сортировка слиянием (External merge sort)

Описание

Внешняя сортировка – это сортировка данных, которые расположены на внешних устройствах и не вмещающихся в оперативную память.

Основным понятием при использовании внешней сортировки является понятие серии. **Серия (упорядоченный отрезок)** – это последовательность элементов, которая упорядочена по ключу. Максимальное количество серий в файле N (все элементы не упорядочены). Минимальное количество серий одна (все элементы упорядочены).

В основе большинства методов внешних сортировок лежит процедура слияния и процедура распределения. **Слияние** – это процесс объединения двух (или более) упорядоченных серий в одну упорядоченную последовательность при помощи циклического выбора элементов, доступных в данный момент. **Распределение** – это процесс разделения упорядоченных серий на два и несколько вспомогательных файла.

Фаза – это действия по однократной обработке всей последовательности элементов. **Двухфазная сортировка** – это сортировка, в которой отдельно реализуется две фазы: распределение и слияние. **Однофазная сортировка** – это сортировка, в которой объединены фазы распределения и слияния в одну.

Двухпутевым слиянием называется сортировка, в которой данные распределяются на два вспомогательных файла. **Многопутевым слиянием** называется сортировка, в которой данные распределяются на N ($N > 2$) вспомогательных файлов.

Сортировка простым слиянием (прямое слияние, Direct merge sort)

В данном алгоритме длина серий фиксируется на каждом шаге. В исходном файле все серии имеют длину 1, после первого шага она равна 2, после второго – 4, после третьего – 8, после k -го шага – 2^k .

Алгоритм:

1. Исходный файл f разбивается на два вспомогательных файла $f1$ и $f2$.
2. Вспомогательные файлы $f1$ и $f2$ сливаются в файл f , при этом одиночные элементы образуют упорядоченные пары.
3. Полученный файл f вновь обрабатывается, как указано в шагах 1 и 2. При этом упорядоченные пары переходят в упорядоченные четверки.
4. Повторяя шаги, сливаем четверки в восьмерки и т.д., каждый раз удваивая длину слитых последовательностей до тех пор, пока не будет упорядочен целиком весь файл.

После выполнения i проходов получаем два файла, состоящих из серий длины 2^i . Окончание процесса происходит при выполнении условия $2^i \geq n$. Следовательно, процесс сортировки простым слиянием требует порядка $O(\log(n))$ проходов по данным.

Замечание: При использовании метода прямого слияния не принимается во внимание то, что исходный файл может быть частично отсортированным, т.е. может содержать упорядоченные подпоследовательности записей.

Исходный файл f : 5 7 3 2 8 4 1

	<i>Распределение</i>	<i>Слияние</i>
1 проход	$f1: 5 \ 3 \ 8 \ 1$ $f2: 7 \ 2 \ 4$	$f: 5 \ 7 \ 2 \ 3 \ 4 \ 8 \ 1$
2 проход	$f1: 5 \ 7 \ 4 \ 8$ $f2: 2 \ 3 \ 1$	$f: 2 \ 3 \ 5 \ 7 \ 1 \ 4 \ 8$
3 проход	$f1: 2 \ 3 \ 5 \ 7$ $f2: 1 \ 4 \ 8$	$f: 1 \ 2 \ 3 \ 4 \ 5 \ 7 \ 8$

Исходный и вспомогательные файлы будут $O(\log(n))$ раз прочитаны и столько же

раз записаны.

Сложность: $O(n * \log(n))$.

Сортировка естественным слиянием (Natural merge sort)

В сравнении с методом прямого слияния, сортировка обладает некоторым преимуществом: учитывается тот факт, что могут содержаться упорядоченные подпоследовательности. То есть, **длина серии** не ограничивается, а определяется количеством элементов в уже упорядоченных подпоследовательностях, выделяемых на каждом проходе.

Алгоритм:

1. Исходный файл f разбивается на два вспомогательных файла $f1$ и $f2$.
Распределение происходит следующим образом: поочередно считываются записи a_i исходной последовательности (неупорядоченной) таким образом, что если значения ключей соседних записей удовлетворяют условию $f(a_i) \leq f(a_{i+1})$, то они записываются в первый вспомогательный файл $f1$. Как только встречаются $f(a_i) > f(a_{i+1})$, то записи a_{i+1} копируются во второй вспомогательный файл $f2$. Процедура повторяется до тех пор, пока все записи исходной последовательности не будут распределены по файлам.
2. Вспомогательные файлы $f1$ и $f2$ сливаются в файл f , при этом серии образуют упорядоченные последовательности.
3. Полученный файл f вновь обрабатывается, как указано в шагах 1 и 2.
4. Повторяя шаги, сливаем упорядоченные серии до тех пор, пока не будет упорядочен целиком весь файл.

Естественное слияние, у которого после фазы распределения количество серий во вспомогательных файлах отличается друг от друга не более чем на единицу, называется **сбалансированным слиянием**, в противном случае – **несбалансированное слияние**.

a : 15 24 32 ‘ 18 45 ‘ 40 51 ‘ 21 29 31 43 ‘ 42 60

b : 15 24 32 ‘ 40 51 ‘ 42 60

c : 18 45 ‘ 21 29 31 43

b : 15 24 32 40 51 ‘ 42 60

c : 18 45 ‘ 21 29 31 43.

a : 15 18 24 32 40 45 51 ‘ 21 29 31 42 43 60.

b : 15 18 24 32 40 45 51

c : 21 29 31 42 43 60.

a : 15 18 21 24 29 31 32 40 42 43 45 51 60.

Таким образом, число чтений или перезаписей файлов при использовании метода естественного слияния будет не хуже, чем при применении метода простого слияния, а в среднем – даже лучше. Но в этом методе увеличивается число сравнений за счет тех, которые требуются для распознавания концов серий. Помимо этого, максимальный размер вспомогательных файлов может быть близок к размеру исходного файла, так как длина серий может быть произвольной.

Сортировка многопутевым слиянием (Multi-Way merge sort)

Данный метод сортировки является модификацией метода естественного слияния. Временные затраты на любую сортировку последовательностей пропорциональны числу требуемых проходов, так как при каждом проходе копируются все данные. Чтобы уменьшить число этих проходов, серии распределяют на последовательности, число которых больше 2-х.

Слияние P серий, поровну распределены в M последовательностей, дает в результате P / M серий. Второй проход уменьшить это число до P / M^2 , третий – до P / M^3 и т.д. Поэтому общее число проходов многопутевого слияния будет равно $\log m (K)$, где K – число элементов в последовательности. Итак, в этом методе, по сравнению с методом естественного слияния, добавляются M путей распределения и слияния последовательностей.

Рассмотрим в качестве примера сортировку трехпутевым слиянием следующей последовательности:

f : 57 24 88 13 19 17 96 37 42 15 21 35 23 10 53 49 33 58 16 72.

f_1 : 57 ‘ 17 96 ‘ 23 ‘ 16 72

f_2 : 24 88 ‘ 37 42 ‘ 10 53 ‘ 49

f_3 : 13 19 ‘ 15 21 35 ‘ 33 58

f_1 : 17 96 ‘ 23 ‘ 16 72

f_4 : 13 19 24 57 88

f_2 : 37 42 ‘ 10 53 ‘ 49

f_5 : .

f_3 : 15 21 35 ‘ 33 58

f_6 :

f_1 : 23 ‘ 16 72

f_4 : 13 19 24 57 88

$f_1: 10 \ 53 \ 49 \ 16 \ 72 \ 13 \ 19 \ 24 \ 57 \ 88$

$f_2: 10 \ 53 \ 49 \quad f_5: 15 \ 17 \ 21 \ 35 \ 37 \ 42 \ 96.$

$f_3: 33 \ 58 \quad f_6:$

$f_1: 16 \ 72 \quad f_4: 13 \ 19 \ 24 \ 57 \ 88$

$f_2: 49 \quad f_5: 15 \ 17 \ 21 \ 35 \ 37 \ 42 \ 96.$

$f_3: \quad f_6: 10 \ 23 \ 33 \ 53 \ 58$

$f_1: \quad f_4: 13 \ 19 \ 24 \ 57 \ 88 \ 16 \ 49 \ 72$

$f_2: \quad f_5: 15 \ 17 \ 21 \ 35 \ 37 \ 42 \ 96.$

$f_3: \quad f_6: 10 \ 23 \ 33 \ 53 \ 58$

$f_4: 13 \ 19 \ 24 \ 57 \ 88 \ 16 \ 49 \ 72 \quad f_1:$

$f_5: 15 \ 17 \ 21 \ 35 \ 37 \ 42 \ 96 \quad f_2:$

$f_6: 10 \ 23 \ 33 \ 53 \ 58 \quad f_3:$

$f_4: 16 \ 49 \ 72 \ f_1: 10 \ 13 \ 15 \ 17 \ 19 \ 21 \ 23 \ 24 \ 33 \ 35 \ 37 \ 42 \ 53 \ 57 \ 58 \ 88 \ 96$

$f_5: \quad f_2:$

$f_6: \quad f_3:$

$f_4: \quad f_1: 10 \ 13 \ 15 \ 17 \ 19 \ 21 \ 23 \ 24 \ 33 \ 35 \ 37 \ 42 \ 53 \ 57 \ 58 \ 88 \ 96$

$f_5: \quad f_2: 16 \ 49 \ 72$

$f_6: \quad f_3:$

$f_1: \quad f_4: 10 \ 13 \ 15 \ 16 \ 17 \ 19 \ 21 \ 23 \ 24 \ 33 \ 35 \ 37 \ 42 \ 49 \ 53 \ 57 \ 58 \ 72 \ 88 \ 96$

$f_2: \quad f_5:$

$f_3: \quad f_6:$

Рассмотренный пример показывает, что алгоритм многопутевого (естественного) слияния работает эффективнее, чем рассмотренные ранее алгоритмы прямого и естественного слияния. Так, при трехпутевом слиянии для сортировки указанной последовательности потребовалось только три прохода, в то время как для естественного и прямого слияния потребовалось бы четыре и пять проходов соответственно.

Дополнительно: <https://intuit.ru/studies/courses/648/504/lecture/11473>,
https://studref.com/701956/informatika/estestvennoe_sliyanie,
<https://studfile.net/preview/930712/page:7>,
https://studref.com/701957/informatika/mnogoputevaya_sortirovka,
<https://www.geeksforgeeks.org/external-sorting>, <https://josef.codes/sorting-really-large-files-with-c-sharp>

Реализация (C# пример)

Сортировка прямым слиянием

```
public class DirectMergeSorter
{
    public string InputFilePath { get; set; }

    public string OutputFilePath { get; set; } = "sorted.csv";

    public int SortKey { get; set; }

    private long _seriesLength;

    private long _countSegments;

    private const string AuxiliaryFilePathA = "A.csv";

    private const string AuxiliaryFilePathB = "B.csv";

    private readonly Func<double, double, bool> _ascending = (x, y) => x < y;

    private readonly Func<double, double, bool> _descending = (x, y) => x > y;

    public DirectMergeSorter(string filePath = "unsorted.csv", int sortKey = 1)
    {
        InputFilePath = filePath;
        SortKey = sortKey;
        _seriesLength = 1;
    }

    public void Sort() => Sort(_ascending);

    public void SortDescending() => Sort(_descending);

    private void Sort(Func<double, double, bool> order)
    {
        var index = 1;
        while (true)
        {
            SplitToFiles();
        }
    }
}
```

```

        if (_countSegments == 1) break;

        Merge(order);
    }
}

private static int GetCountLinesFile(string filePath)
{
    using var sr = new StreamReader(filePath);
    var count = 0;
    while ((sr.ReadLine()) != null)
    {
        count++;
    }

    return count;
}

private void SplitToFiles()
{
    _countSegments = 1;

    using var sr = _seriesLength == 1 ? new StreamReader(InputFilePath)

    using var swA = new StreamWriter(AuxiliaryFilePathA);
    using var swB = new StreamWriter(AuxiliaryFilePathB);

    var counter = 0L;
    var isFirstFile = true;

    var length = GetCountLinesFile(_seriesLength == 1 ? InputFilePath :
    var position = 0L;

    while (position != length)
    {
        if (counter == _seriesLength)
        {
            isFirstFile = !isFirstFile;
            counter = 0;
            _countSegments++;
        }

        var str = sr.ReadLine();

        position++;

        if (isFirstFile)
        {
            swA.WriteLine(str);
        }
        else

```

```

        {
            swB.WriteLine(str);
        }

        counter++;
    }
}

private void Merge(Func<double, double, bool> comparer)
{
    using var srA = new StreamReader(AuxiliaryFilePathA);
    using var srB = new StreamReader(AuxiliaryFilePathB);

    using var sw = new StreamWriter(OutputFilePath);

    var counterA = _seriesLength;
    var counterB = _seriesLength;

    var strA = "";
    var strB = "";

    var isPickedA = false;
    var isPickedB = false;
    var isEndA = false;
    var isEndB = false;

    var lengthA = GetCountLinesFile(AuxiliaryFilePathA);
    var lengthB = GetCountLinesFile(AuxiliaryFilePathB);

    var positionA = 0L;
    var positionB = 0L;

    while (!isEndA || !isEndB)
    {
        if (counterA == 0 && counterB == 0)
        {
            counterA = _seriesLength;
            counterB = _seriesLength;
        }

        if (positionA != lengthA)
        {
            if (counterA > 0 && !isPickedA)
            {
                strA = srA.ReadLine();
                positionA++;
                isPickedA = true;
            }
        }
        else
        {

```

```

        isEndA = true;
    }

    if (positionB != lengthB)
    {
        if (counterB > 0 && !isPickedB)
        {
            strB = srB.ReadLine();
            positionB++;
            isPickedB = true;
        }
    }
    else
    {
        isEndB = true;
    }

    if (isPickedA)
    {
        if (isPickedB)
        {
            if (comparer(double.Parse(strA.Split(";")[SortKey], Cul
            {
                sw.WriteLine(strA);
                counterA--;
                isPickedA = false;
            }
            else
            {
                sw.WriteLine(strB);
                counterB--;
                isPickedB = false;
            }
        }
        else
        {
            sw.WriteLine(strA);
            counterA--;
            isPickedA = false;
        }
    }
    else if (isPickedB)
    {
        sw.WriteLine(strB);
        counterB--;
        isPickedB = false;
    }
}

_seriesLength *= 2;
}

```

```
}
```

Сортировка естественным слиянием

```
public class NaturalMergeSorter
{
    public string InputFilePath { get; set; }

    public string OutputFilePath { get; set; } = "sorted.csv";

    public int SortKey { get; set; }

    private bool _isPassageFirst = true;

    private long _countSegments;

    private const string AuxiliaryFilePathA = "A.csv";

    private const string AuxiliaryFilePathB = "B.csv";

    private readonly Func<double, double, bool> _ascending = (x, y) => x < y;

    private readonly Func<double, double, bool> _descending = (x, y) => x > y;

    public NaturalMergeSorter(string filePath = "unsorted.csv", int sortKey
    {
        InputFilePath = filePath;
        SortKey = sortKey;
    }

    public void Sort() => Sort(_ascending);

    public void SortDescending() => Sort(_descending);

    private void Sort(Func<double, double, bool> order)
    {
        var index = 1;
        while (true)
        {
            SplitToFiles(order);

            if (_countSegments == 1) break;

            Merge(order);
        }
    }

    private static int GetCountLinesFile(string filePath)
    {
```

```

        using var sr = new StreamReader(filePath);
        var count = 0;
        while (sr.ReadLine() != null)
        {
            count++;
        }

        return count;
    }

    private void SplitToFiles(Func<double, double, bool> comparer)
    {
        _countSegments = 1;

        using var sr = _isPassageFirst ? new StreamReader(InputFilePath) :

        using var swA = new StreamWriter(AuxiliaryFilePathA);
        using var swB = new StreamWriter(AuxiliaryFilePathB);

        var isStart = false;
        var isFirstFile = true;

        var length = GetCountLinesFile(_isPassageFirst ? InputFilePath : Ou
        _isPassageFirst = false;
        var position = 0L;

        var str = "";
        var nextStr = "";

        if (length == 1)
        {
            swA.WriteLine(sr.ReadLine());
            return;
        }

        while (position != length)
        {
            if (!isStart)
            {
                str = sr.ReadLine();
                position++;

                swA.WriteLine(str);
                isStart = true;
            }

            nextStr = sr.ReadLine();

            position++;

            if (comparer(double.Parse(nextStr.Split(";")[SortKey], CultureI

```

```

        {
            isFirstFile = !isFirstFile;
            _countSegments++;
        }

        if (isFirstFile)
        {
            swA.WriteLine(nextStr);
        }
        else
        {
            swB.WriteLine(nextStr);
        }

        str = nextStr;
    }
}

private void Merge(Func<double, double, bool> comparer)
{
    using var srA = new StreamReader(AuxiliaryFilePathA);
    using var srB = new StreamReader(AuxiliaryFilePathB);

    using var sw = new StreamWriter(OutputFilePath);

    var strA = "";
    var strB = "";

    var isPickedA = false;
    var isPickedB = false;
    var isEndA = false;
    var isEndB = false;

    var lengthA = GetCountLinesFile(AuxiliaryFilePathA);
    var lengthB = GetCountLinesFile(AuxiliaryFilePathB);

    var positionA = 0L;
    var positionB = 0L;

    while (!isEndA || !isEndB || isPickedA || isPickedB)
    {
        isEndA = positionA == lengthA;
        isEndB = positionB == lengthB;

        if (!isEndA && !isPickedA)
        {
            strA = srA.ReadLine();

            positionA++;
            isPickedA = true;
        }
    }
}

```



```

        if (!isEndB && !isPickedB)
        {
            strB = srB.ReadLine();

            positionB++;
            isPickedB = true;
        }

        if (isPickedA)
        {
            if (isPickedB)
            {
                if (comparer(double.Parse(strA.Split(";")[SortKey], Cul
                    double.Parse(strB.Split(";")[SortKey], CultureI
                {
                    sw.WriteLine(strA);
                    isPickedA = false;
                }
                else
                {
                    sw.WriteLine(strB);
                    isPickedB = false;
                }
            }
            else
            {
                sw.WriteLine(strA);
                isPickedA = false;
            }
        } else if (isPickedB)
        {
            sw.WriteLine(strB);
            isPickedB = false;
        }
    }
}
}

```

Сортировка многопутевым (прямым) слиянием

```

public class MultiWayMergeSorter
{
    private class HeadIndexPair
    {
        public string? Head { get; }
        public int Index { get; }

        public HeadIndexPair(string? head, int index)
    }
}

```



```

    {
        Head = head;
        Index = index;
    }
}

public string InputFilePath { get; set; }

public string OutputFilePath { get; set; } = "sorted.csv";

public int SortKey { get; set; }

public int CountChunkRows { get; set; }

public string ColumnSeparator { get; set; }

private const string TmpFilePrefix = "tmpFile";
private int _numChunk;

public MultiWayMergeSorter(string inputFilePath = "unsorted.csv", int c
{
    InputFilePath = inputFilePath;
    CountChunkRows = countChunkRows;
    SortKey = sortKey;
    ColumnSeparator = columnSeparator;
}

public void Sort()
{
    using var sr = new StreamReader(InputFilePath);
    var cnt = 0;

    var chunk = new string[CountChunkRows];

    string? line;
    while ((line = sr.ReadLine()) != null)
    {
        chunk[cnt] = line;
        cnt++;
        if (cnt % CountChunkRows == 0)
        {
            SortAndSaveChunk(chunk, TmpFilePrefix + _numChunk);
            cnt = 0;
            _numChunk++;
        }
    }

    if (cnt != 0)
    {
        SortAndSaveChunk(chunk, TmpFilePrefix + _numChunk);
        _numChunk++;
    }
}

```

```

    }

    var readers = new StreamReader[_numChunk];

    var heads = new PriorityQueue<HeadIndexPair, HeadIndexPair>(Compare

    for (var i = 0; i < _numChunk; i++)
    {
        var strRead = new StreamReader(TmpFilePrefix + i);
        readers[i] = strRead;
    }

    using (var streamOut = new StreamWriter(OutputFilePath))
    {
        for (var i = 0; i < _numChunk; i++)
        {
            heads.Enqueue(new HeadIndexPair(readers[i].ReadLine(), i),

        }

        while (true)
        {
            var minH = heads.Count > 0 ? heads.Dequeue() : null;
            if (null == minH) break;

            streamOut.WriteLine(minH.Head);

            if ((line = readers[minH.Index].ReadLine()) != null)
                heads.Enqueue(new HeadIndexPair(line, minH.Index), new |

        }

        for (var i = 0; i < _numChunk; i++)
        {
            readers[i].Close();
        }

    }

    sr.Close();
}

private void SortAndSaveChunk(string?[] chunk, string filename)
{
    Array.Sort(chunk, Compare);
    using var sw = new StreamWriter(filename);
    foreach (var t in chunk)
    {
        if (t != null) sw.WriteLine(t);
    }

    sw.Close();
}

```

```
private string ExtractCol(string line)
{
    var columns = line.Split(ColumnSeparator);
    return columns[SortKey];
}

private int Compare(string? a, string? b)
{
    if (a == null && b == null) return 0;
    if (a == null) return 1;
    if (b == null) return -1;
    return string.Compare(ExtractCol(a), ExtractCol(b), StringComparison...)
}
}
```

Билет 9: Сортировка с помощью двоичного дерева (Tree Sort)

Описание

Алгоритм сортировки, основанный на структуре данных [двоичное дерево поиска](#). Сначала он создаёт двоичное дерево поиска из элементов входного списка или массива, а затем выполняет обход созданного двоичного дерева поиска по порядку, чтобы получить элементы в отсортированном порядке.

Алгоритм:

1. Возьмите элементы, введенные в массив.
2. Создайте двоичное дерево поиска, вставив элементы данных из массива в двоичное дерево поиска.
3. Выполните обход дерева по порядку, чтобы получить элементы в отсортированном порядке.

Сложность:

- Средний случай: $O(n * \log(n))$, добавление одного элемента в дерево двоичного поиска в среднем занимает $O(\log(n))$ времени. Следовательно, добавление n элементов займет $O(n * \log(n))$ времени;
- Худший случай: $O(n^2)$, может быть улучшена с помощью самобалансирующегося двоичного дерева поиска;

Вспомогательное пространство: $O(n)$.

Двоичное дерево поиска

Структура данных двоичного дерева на основе узлов, которая обладает следующими свойствами:

- *Левое поддерево узла содержит только узлы с ключами меньше, чем ключ узла.*
- *Правое поддерево узла содержит только узлы с ключами больше, чем ключ узла.*
- *Каждое из левого и правого поддеревьев также должно быть бинарным деревом поиска.*

Поиск

Алгоритм:

1. Начните с корня.
2. Сравните искомый элемент с корнем, если он меньше корня, то рекурсивно вызовите левое поддерево, иначе рекурсивно вызовите правое поддерево.
3. Если элемент для поиска найден где угодно, верните true, иначе верните false.

Временная сложность: $O(h)$, **Пространственная сложность:** $O(h)$, где h — высота бинарного дерева поиска.

Вставка

Алгоритм:

1. Начните с корня.
2. Сравните вставляемый элемент с корнем, если он меньше корня, то рекурсивно вызовите левое поддерево, иначе рекурсивно вызовите правое поддерево.
3. Достигнув конца, просто вставьте этот узел слева (если он меньше текущего) или справа.

Временная сложность: $O(h)$, где h — высота бинарного дерева поиска,
Вспомогательное пространство: $O(1)$.

Дополнительно: <https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion>, <https://www.geeksforgeeks.org/tree-sort>,
<https://www.programiz.com/dsa/binary-tree>,
<https://intuit.ru/studies/courses/648/504/lecture/11472>

Реализация (C# пример)



```
public class Node<T> where T : IComparable
{
    public T Value { get; set; }

    public Node<T>? Left { get; private set; }

    public Node<T>? Right { get; private set; }

    public Node(T value)
    {
        Value = value;
    }

    public void Insert(T value)
    {
        if (Value.CompareTo(value) > 0) // Сортировка по возрастанию
        {
            if (Left == null)
            {
                Left = new Node<T>(value);
            }
            else
            {
                Left.Insert(value);
            }
        }
        else
        {
            if (Right == null)
            {
                Right = new Node<T>(value);
            }
            else
            {
                Right.Insert(value);
            }
        }
    }

    public IList<T> ToList(IList<T>? collection = null)
    {
        var index = 0;
        return ToList(ref index, collection);
    }

    private IList<T> ToList(ref int index, IList<T>? collection = null)
    {
        collection ??= new List<T>();

        Left?.ToList(ref index, collection);
```

```

        collection[index++] = Value;

        Right?.ToList(ref index, collection);

        return collection;
    }
}

public static IList<T> TreeSort<T>(this IList<T> collection) where T : ICom
{
    if (collection.Count > 0)
    {
        var node = new Node<T>(collection[0]);
        for (var i = 1; i < collection.Count; i++)
        {
            node.Insert(collection[i]);
        }

        node.ToList(collection);
    }

    return collection;
}

```

Билет 10: Поразрядная сортировка (Radix Sort)

Описание

Алгоритм, который сортирует элементы, сначала группируя отдельные цифры одного и того-же разряда. Затем сортирует элементы в порядке возрастания/убывания.

Идея сортировки по основанию состоит в том, чтобы выполнять сортировку по цифрам, начиная с младшей значащей цифры и заканчивая старшей значащей цифрой. Сортировка по основанию использует [сортировку подсчетом](#) в качестве подпрограммы для сортировки.

Алгоритм:

1. Найдите самый большой элемент в массиве, т.е. *max*. Пусть *X* будет количество цифр в *max*. *X* вычисляется, потому что мы должны пройти все значимые места всех элементов.
2. Теперь пройдите по каждому значимому месту одно за другим. Отсортируйте элементы по цифрам разряда.

Сложность:

- Лучший случай: $O(n + k)$;
- Средний случай: $O(n + k)$;
- Худший случай: $O(n + k)$;

Вспомогательное пространство: $O(max)$;

Сортировка подсчётом

Алгоритм сортировки, который сортирует элементы массива, подсчитывая количество вхождений каждого уникального элемента в массиве. Счетчик хранится во вспомогательном массиве, а сортировка выполняется путем отображения счетчика как индекса вспомогательного массива.

Алгоритм:

1. Найдите максимальный элемент (пусть это будет *max*) из заданного массива.
2. Инициализировать массив длины *max + 1*. Этот массив используется для хранения количества элементов в массиве.
3. Сохраните количество каждого элемента в соответствующем индексе в *count* массиве.
4. Храните кумулятивную сумму элементов массива *count*. Это помогает поместить элементы в правильный индекс отсортированного массива.
5. Найдите индекс каждого элемента исходного массива в массиве *count*. Это дает кумулятивный счет.
6. После размещения каждого элемента в правильном месте уменьшите его количество на единицу.

Сложность:

- Лучший случай: $O(n + k)$;
- Средний случай: $O(n + k)$;
- Худший случай: $O(n + k)$;

Вспомогательное пространство: $O(max)$;

Дополнительно: <https://www.geeksforgeeks.org/radix-sort>, <https://www.programiz.com/dsa/radix-sort>, <https://www.programiz.com/dsa/counting-sort>

Реализация (C# пример)

```
public static IList<int> RadixSort(this IList<int> collection)
{
```




```

var maxValue = collection.Max();

for (var exponent = 1; maxValue / exponent > 0; exponent *= 10)
{
    collection.CountingSort(exponent);
}

return collection;
}

public static IList<int> CountingSort(this IList<int> collection, int exponent)
{
    var outputArr = new int[collection.Count];
    var countArr = new int[10];

    foreach (var item in collection) // Считаем количество вхождений каждого
    {
        countArr[(item / exponent) % 10]++;
    }

    for (var i = 1; i < 10; i++) // Сохраняем фактические позиции элементов
    {
        countArr[i] += countArr[i - 1];
    }

    for (var i = collection.Count - 1; i >= 0; i--) // Устанавливаем элемент
    {
        outputArr[countArr[(collection[i] / exponent) % 10] - 1] = collection[i];
        countArr[(collection[i] / exponent) % 10]--;
    }

    for (var i = 0; i < collection.Count; i++) // Копируем элементы в исходный массив
    {
        collection[i] = outputArr[i];
    }

    return collection;
}

```

Билет 11: Хэш-таблицы с разрешением коллизий методом цепочек

Описание

В хеш-таблице новый индекс обрабатывается с помощью ключей. Элемент, соответствующий этому ключу, сохраняется в индекс. Этот процесс, называется хешированием. Хорошая хеш-функция предполагает достаточно быстрое

вычисление, сводит к минимуму число коллизий.

Идей цепочек состоит в том, чтобы реализовать массив, как связный список, называемый цепочкой. Это один из самых популярных и часто используемых методов обработки коллизий. Когда возникает ситуация, что несколько элементов хэшируются в один и тот же индекс слота, затем эти элементы вставляются в односвязный список, известный как цепочка. Теперь мы можем использовать ключ K для поиска в связном списке, просто просматривая его линейно. Если внутренний ключ для какой-то записи совпадёт с ключом K , это будет означать, что мы нашли нашу запись. В случае, если мы достигли конца связного списка, но не нашли нашу запись, то это будет означать, что запись не существует. **Вывод:** если в отдельной цепочке два разных элемента имеют одинаковое значение хеш-функции, мы сохраняем оба элемента в одном и том же связном списке один за другим.

Производительность цепочек: Производительность хеширования можно оценить в предположении, что каждый ключ с одинаковой вероятностью будет хэширован в любой слот таблицы (просто равномерное хеширование).

- m - количество слотов в хеш-таблице;
- n - количество вставленных значений в хеш-таблицу;
- Коэффициент загрузки: $\alpha = n / m$;
- Ожидаемое время поиска: $O(1 + \alpha)$;
- Ожидаемое время удаления: $O(1 + \alpha)$;
- Время на вставку: $O(1)$;
- Сложность поиска, вставки и удаления равна $O(1)$, если $\alpha = O(1)$.

Дополнительно: <https://www.geeksforgeeks.org/separate-chaining-collision-handling-technique-in-hashing>, <https://www.programiz.com/dsa/hash-table>, <https://iq.opengenus.org/time-complexity-of-hash-table>

Реализация (C# пример)

```
/// <summary> Словарь: хеш-таблица, разрешение коллизий методом цепочек </summary>
/// <typeparam name="TKey"> Тип ключа </typeparam>
/// <typeparam name="TValue"> Тип значения </typeparam>
public class Dictionary<TKey, TValue> : IEnumerable<KeyValuePair<TKey, TValue>
{
    private readonly int _size;

    public double FillFactor => (double)Count / _size;

    public int MaxLengthChain => _items.Max(x => x?.Count ?? 0);
```

```

public int MinLengthChain => _items.Min(x => x?.Count ?? 0);

public IEnumerable<int> LengthsChains => _items.Select(x => x?.Count ??

public int Count { get; private set; }

private readonly LinkedList<KeyValuePair<TKey, TValue?>>[] _items;

public Dictionary(int size = 1000)
{
    if (!IsSizeCorrect(size)) throw new AggregateException(nameof(size)
        _size = size;
        _items = new LinkedList<KeyValuePair<TKey, TValue?>>[size];
}

public void Add(TKey key, TValue value)
{
    if (key == null) throw new ArgumentNullException(nameof(key));
    var item = new KeyValuePair<TKey, TValue?>(key, value);
    Insert(item);
}

protected void Insert(KeyValuePair<TKey, TValue?> item)
{
    var position = GetListPosition(item.Key);
    var linkedList = GetLinkedList(position);

    foreach (var pair in linkedList)
    {
        if (pair.Key != null && pair.Key.Equals(item.Key))
            throw new ArgumentException("Элемент по указанному ключу уж

    linkedList.AddLast(item);
    Count++;
}

protected bool IsSizeCorrect(int size)
{
    return size > 0;
}

public bool Remove(TKey key)
{
    var position = GetListPosition(key);
    var linkedList = GetLinkedList(position);
    var itemFound = false;
    var foundItem = default(KeyValuePair<TKey, TValue?>);
    foreach (var item in linkedList)
    {
        if (item.Key != null && item.Key.Equals(key))

```

```

        {
            itemFound = true;
            foundItem = item;
        }
    }

    if (itemFound)
    {
        linkedList.Remove(foundItem);
        Count--;
    }
    return itemFound;
}

public bool ContainsKey(TKey key)
{
    if (key == null) throw new ArgumentNullException(nameof(key));
    var position = GetListPosition(key);
    var linkedList = GetLinkedList(position);

    var foundItem = false;
    foreach (var item in linkedList)
    {
        if (item.Key != null && item.Key.Equals(key))
        {
            foundItem = true;
            break;
        }
    }

    return foundItem;
}

public TValue? GetValue(TKey key)
{
    if (key == null) throw new ArgumentNullException(nameof(key));
    var position = GetListPosition(key);
    var linkedList = GetLinkedList(position);
    foreach (var item in linkedList)
    {
        if (item.Key != null && item.Key.Equals(key)) return item.Value
    }

    return default;
}

public bool TryGetValue(TKey key, out TValue value)
{
    var t = GetValue(key);
    value = t;
    return t != null;
}

```

```

}

protected int GetListPosition(TKey key)
{

    return Math.Abs(key.GetHashCode() % _size); // Метод деления

    // Метод умножения
    // var goldenRatioConst = (Math.Sqrt(5) - 1) / 2;
    // return (int) Math.Abs(_size * (key.GetHashCode() * goldenRatioCo
}

protected LinkedList<KeyValuePair<TKey, TValue?>> GetLinkedList(int pos
{
    var linkedList = _items[position];
    if (linkedList == null)
    {
        linkedList = new LinkedList<KeyValuePair<TKey, TValue?>>();
        _items[position] = linkedList;
    }

    return linkedList;
}

public void Clear()
{
    if (Count > 0)
    {
        for (var i = 0; i < _items.Length; i++)
        {
            _items[i] = null;
        }
    }
}

public IEnumerator<KeyValuePair<TKey, TValue?>> GetEnumerator()
{
    foreach (var linkedList in _items)
    {
        if (linkedList != null)
        {
            foreach (var keyValuePair in linkedList)
            {
                yield return keyValuePair;
            }
        }
    }
}

IEnumerator IEnumerable.GetEnumerator()
{

```

```
        return GetEnumerator();  
    }  
}
```

Билет 12: Хэш-таблицы с разрешением коллизий методом открытой адресации

Описание

Подобно методу цепочек, открытая адресация является методом обработки коллизий. В открытой адресации все элементы хранятся в самой хеш-таблице. Таким образом, в любой момент размер таблицы должен быть больше или равен общему количеству ключей (обратите внимание, что мы можем увеличить размер таблицы, скопировав старые данные, если это необходимо). Этот подход также известен как закрытое хеширование. Вся эта процедура основана на исследованиях: линейное, квадратичное, двойное.

Преимущества перед методом цепочек:

- Открытая адресация повышает скорость кэширования, поскольку все данные хранятся в хеш-таблице.
- Он правильно использует свои пустые индексы и повышает эффективность использования памяти.
- Поскольку не используется связанный список или указатель, производительность выше, чем при цепочке или открытом хешировании.

Операции:

- добавление элемента — продолжается поиск, пока не будет найден пустой слот, как только найден, вставляем k ;
- поиск элемента — продолжать поиск пока ключ слота не станет равным k или пока не будет достигнут пустой слот;
- удаление — есть нюансы, если мы просто удалим ключ, то поиск может не получиться, поэтому слоты удалённых ключей помечаются как удалённые, возможность вставить элемент в удалённый слот останется, но поиск на месте удалённого слота останавливаться не будет;

Производительность открытой адресации: Подобно цепочке, производительность хеширования можно оценить, исходя из предположения, что каждый ключ с одинаковой вероятностью будет хэширован в любой слот таблицы (простое равномерное хеширование).

- m - количество слотов в хеш-таблице;
- n - количество вставленных значений в хеш-таблицу;
- Коэффициент загрузки: $\alpha = n / m (< 1)$;
- Сложность поиска, вставки и удаления равна $O(1 / (1 + \alpha))$.

Дополнительно: <https://www.geeksforgeeks.org/open-addressing-collision-handling-technique-in-hashing>, <https://www.programiz.com/dsa/hash-table>, <https://iq.opengenus.org/time-complexity-of-hash-table>

Реализация (C# пример)

```

/// <summary> Хеш-таблица, разрешение коллизий методом открытой адресации </summary>
/// <typeparam name="TKey"> Тип ключа </typeparam>
/// <typeparam name="TValue"> Тип значения </typeparam>
public class Hashtable<TKey, TValue> : IEnumerable<KeyValuePair<TKey, TValue>>
{
    private readonly int _size;

    private readonly KeyValuePair<TKey, TValue?>[] _items;

    private readonly bool[] _removed;

    private static readonly Func<Func<object, int, int>, object, int, int,
        (f, key, sizeHashTable, index) => (f(key, sizeHashTable) + index) %

    private static readonly Func<Func<object, int, int>, object, int, int,
        (f, key, sizeHashTable, index) => (f(key, sizeHashTable) + (int)Mat

    private static readonly Func<Func<object, int, int>, Func<object, int,
        (f1, f2, key, sizeHashTable, index) => (f1(key, sizeHashTable) + in

    public int Count { get; private set; }

    public int MaxClusterLength
    {
        get
        {
            var max = 0;
            var current = 0;
            foreach (var item in _items)
            {
                if (!item.Equals(default(KeyValuePair<TKey, TValue?>)))
                {
                    current++;
                }
                else
                {
                    max = Math.Max(max, current);
                }
            }
        }
    }
}

```

```

        current = 0;
    }
}

return Math.Max(max, current);
}
}

public Hashtable(int size = 1000)
{
    if (!IsSizeCorrect(size)) throw new AggregateException(nameof(size),
        "Размер хеш-таблицы должен быть больше 0");
    _size = size;
    _items = new KeyValuePair<TKey, TValue?>[_size];
    _removed = new bool[_size];
}

protected bool CheckOpenSpace()
{
    var isOpen = false;
    for (var i = 0; i < _size; i++)
    {
        if (_items[i].Equals(default(KeyValuePair<TKey, TValue?>))) isOpen = true;
    }

    return isOpen;
}

protected bool IsSizeCorrect(int size)
{
    return size > 0;
}

protected bool CheckUniqueKey(TKey key)
{
    foreach (var item in _items)
    {
        if (item.Key != null && item.Key.Equals(key)) return false;
    }

    return true;
}

public void Add(TKey key, TValue value)
{
    if (key == null) throw new ArgumentNullException(nameof(key));

    if (!CheckOpenSpace()) throw new ArgumentOutOfRangeException("Хеш-таблица переполнена");

    if (!CheckUniqueKey(key)) throw new ArgumentException("Элемент по ключу уже существует");

    Insert(key, value);
}

```



```

}

protected void Insert(TKey key, TValue value)
{
    var index = 0;
    var hashCode = GetHash(key, index);

    while (!_items[hashCode].Equals(default(KeyValuePair<TKey, TValue>))
    {
        index++;
        hashCode = GetHash(key, index);
    }

    _items[hashCode] = new KeyValuePair<TKey, TValue?>(key, value);
    _removed[hashCode] = false;
    Count++;
}

protected int GetHash(TKey key, int index)
{
    // Линейный анализ, вспомогательная функция — метод деления.
    return LinearHashing((key, sizeHashTable) => Math.Abs(key.GetHashCode()
}

public TValue? GetValue(TKey key)
{
    if (key == null) throw new ArgumentNullException(nameof(key));

    var index = 0;
    var hashCode = GetHash(key, index);

    while ((!_items[hashCode].Equals(default(KeyValuePair<TKey, TValue>))
    {
        index++;
        hashCode = GetHash(key, index);
    }

    return _items[hashCode].Value;
}

public bool Remove(TKey key)
{
    var index = 0;
    var hashCode = GetHash(key, index);

    while ((!_items[hashCode].Equals(default(KeyValuePair<TKey, TValue>))
    {
        index++;
        hashCode = GetHash(key, index);
    }
}

```

```

        if (_items[hashCode].Equals(default(KeyValuePair<TKey, TValue>)))
        {
            return false;
        }
        else
        {
            _items[hashCode] = default;
            _removed[hashCode] = true;
            Count--;
            return true;
        }
    }

    public IEnumerator<KeyValuePair<TKey, TValue?>> GetEnumerator()
    {
        foreach (var keyValuePair in _items)
        {
            if (!keyValuePair.Equals(default(KeyValuePair<TKey, TValue?>)))
            {
                yield return keyValuePair;
            }
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}

```

Билет 13: ABC сортировка для строк (Allen Beechick Character Sort for string)

Описание

Лексографическая вариация поразрядной MSD (по наибольшей значащей цифре) сортировки. Автор Аллен Бичик выбрал название в честь себя любимого, ABCsort расшифровывается как Allen Beechick Character sort. Сам по себе Бичик примечателен тем, что он не только программист, а ещё и целый магистр богословия.

Для сортировки требуется два вспомогательных массива.

Один из них назовём трекер слов (WT – word tracker), с помощью него мы будем группировать слова, имеющих одинаковые буквы в *i*-м разряде. Для самого первого найденного такого слова в списке заносится значение 0. Для каждого

последующего найденного слова с той же буквой в i -м разряде в трекаре слов отмечается индекс предыдущего слова, соответствующего этому же признаку.

индекс	1	2	3	4	5	6	7	8	9
слово	Carmen	Adela	Beatrix	Abbey	Abigale	Barbara	Camalia	Belinda	Beckie
трекер	0	0	0	2	4	3	1	6	8

Ещё один массив – трекар символов (LT – letter tracker). В нём отмечаются индексы самого первого (или последнего) слова в списке, в котором в соответствующем разряде находится определённый символ. Отталкиваясь от этого слова, с помощью трекера слов восстанавливается цепочка всех остальных лексем, имеющих в i -м разряде соответствующую букву.

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
1	5	9	7																							
2																										
3																										

Создавая и прослеживая подобные цепочки слов, рекурсивно продвигаясь от старших разрядов к младшим, в итоге весьма быстро формируются новые последовательности, упорядоченные в алфавитном порядке. Отсортировав слова на «А», затем сортируется «В», затем «С» и далее по алфавиту.

Трекер слов - Word Tracker - WT																										
1	2	3	4	5	6	7	8	9																		
Carmen	Adela	Beatrix	Abbey	Abigale	Barbara	Camalia	Belinda	Beckie																		
Группируем слова по первым буквам																										
Трекер букв - Letter Tracker - LT																										
	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
1																										
2																										
3																										
Итоговый список																										
1	2	3	4	5	6	7	8	9																		

Сложность: $O(k * n)$, где k — количество обрабатываемых разрядов.

Вспомогательное пространство: $O(n)$.

Дополнительно: <https://habr.com/ru/post/201534/>,
<http://www.aislebyaisle.com/cb/access/sorting/beeckicksort.htm>,
<http://algotlab.valemak.com/radix>

Реализация (C# пример)

```
private static IList<string> ABCSort(this IList<string> collection, int rank)
{
    if (collection.Count < 2) return collection;

    var table = new System.Collections.Generic.Dictionary<char, IList<string>>();
    var listResult = new List<string>();
    var shortWordsCounter = 0;

    foreach (var str in collection)
    {
        if (rank < str.Length)
        {
            if (table.ContainsKey(str[rank]))
            {
                table[str[rank]].Add(str);
            }
            else
            {
                table.Add(str[rank], new List<string> {str});
            }
        }
        else
        {
            listResult.Add(str);
            shortWordsCounter++;
        }
    }

    if (shortWordsCounter == collection.Count) return collection;

    for (var i = 'A'; i <= 'z'; i++)
    {
        if (table.ContainsKey(i))
        {
            foreach (var str in ABCSort(table[i], rank + 1))
            {
                listResult.Add(str);
            }
        }
    }

    return listResult;
}
```

Билет 14: Реализовать стек и базовые операции

работы со стеком, с использованием собственного двусвязного списка

Описание

Стек — это линейная структура данных, которая следует принципу «последним пришел — первым вышел» (LIFO). Это означает, что последний элемент, вставленный в стек, удаляется первым.

Операции:

- *Push* — операция добавления элемента на вершину стека;
- *Pop* — операция извлечения элемента с вершины стека;
- *IsEmpty* — проверка, пуст ли стек;
- *Peek* — получить значение верхнего элемента, не удаляя его;

Сложность операций: $O(1)$.

Дополнительно: <https://www.geeksforgeeks.org/implement-a-stack-using-singly-linked-list>, <https://www.geeksforgeeks.org/c-sharp-stack-with-examples>, <https://www.programiz.com/dsa/stack>, <https://www.programiz.com/csharp-programming/stack>

Реализация (C# пример)

```
public class Stack<T> : IReadOnlyCollection<T>
{
    private class Node<T>
    {
        public T Value { get; }

        public Node<T> Top { get; init; }

        public Node(T value)
        {
            Value = value;
        }
    }

    private Node<T>? _top;

    public int Count { get; private set; }

    public void Push(T element)
    {
        var newNode = new Node<T>(element)
```



```

        {
            Top = _top;
        };
        _top = newNode;
        Count++;
    }

    public T Pop()
    {
        if (IsEmpty()) throw new InvalidOperationException("Стек пуст, нельзя извлечь элемент");

        var elem = _top.Value;
        _top = _top.Top;
        Count--;
        return elem;
    }

    public T Peek()
    {
        if (IsEmpty()) throw new InvalidOperationException("Стек пуст, нельзя посмотреть элемент");

        return _top.Value;
    }

    public bool IsEmpty()
    {
        return _top == null;
    }

    public IEnumerator<T> GetEnumerator()
    {
        var node = _top;
        while (node != null)
        {
            yield return node.Value;
            node = node.Top;
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}

```

Билет 15: Реализовать очередь и базовые операции работы с очередью, с использованием собственного двусвязного списка

Описание

Структура данных, которая следует правилу «первым поступил – первым обслужен» (FIFO) – элемент, который поступает первым, является элементом, который выходит первым.

Операции:

- *Enqueue* — операция добавления элемента в очередь;
- *Dequeue* — операция извлечения элемента из очереди;
- *IsEmpty* — проверка, пуста ли очередь;
- *Peek* — получить значение из начала очереди, не удаляя его;

Сложность операции добавления/удаления: $O(1)$.

Сложность операции извлечения/получения: $O(n)$.

Дополнительно: <https://www.geeksforgeeks.org/queue-data-structure>,
<https://www.programiz.com/dsa/queue>, <https://www.programiz.com/csharp-programming/queue>

Реализация (C# пример)

```
public class Queue<T> : IReadOnlyCollection<T>
{
    public class Node<T>
    {
        public Node(T value)
        {
            Value = value;
        }

        public T Value { get; set; }

        public Node<T> Next { get; set; }
    }

    private Node<T>? _head;

    private Node<T>? _tail;

    public int Count { get; private set; }

    public bool IsEmpty() => Count == 0;

    public void Enqueue(T value)
```



```

{
    var node = new Node<T>(value);
    var tempNode = _tail;
    _tail = node;
    if (Count == 0)
    {
        _head = _tail;
    }
    else
    {
        tempNode.Next = _tail;
    }

    Count++;
}

public T Dequeue()
{
    if (Count == 0) throw new InvalidOperationException();
    var output = _head.Value;
    _head = _head.Next;
    Count--;

    return output;
}

public IEnumerator<T> GetEnumerator()
{
    var node = _head;
    while (node != null)
    {
        yield return node.Value;
        node = node.Next;
    }
}

IEnumerator IEnumerable.GetEnumerator()
{
    return GetEnumerator();
}
}

```

Билет 16: Алгоритм Крускала (поиска минимального остовного дерева)

Описание

Алгоритм поиска минимального остовного дерева, который находит ребро

наименьшего возможного веса, соединяющее любые две вершины графа. Это жадный алгоритм в теории графов, поскольку он находит минимальное остовное дерево для связного взвешенного графа, добавляя рёбра с возрастающей стоимостью на каждом шаге. Это означает, что он находит подмножество ребер, образующих дерево, включающее каждую вершину, где общий вес всех ребер в дереве минимален.

Сложность алгоритма: Сортировка ребер занимает время $O(E * \log(E))$. После сортировки мы перебираем все ребра и применяем алгоритм **Union-Find**. Операции поиска и объединения могут занимать не более $O(\log(V))$ * времени. Таким образом, общая сложность составляет $O(E * \log(E) + E * \log(V))$ времени, где V — количество вершин, E — количество рёбер.

Принцип работы алгоритма:

1. Рёбра графа сортируются в порядке не убывания.
2. Выбираем наименьшее ребро. Проверяем, образует ли он цикл с уже сформированным остовным деревом. Если цикл не формируется, включаем это ребро. В противном случае отказываемся от него.
3. Повторяем шаг 2, пока в связующем дереве не будет $V - 1$ рёбер.

Алгоритм **Union-Find** (используется в шаге 2): это алгоритм, который выполняет две полезные операции: **Найти**: определить, в каком подмножестве находится конкретный элемент. Это можно использовать для определения того, находятся ли два элемента в одном и том же подмножестве. **Объединение**: объединение двух подмножеств в одно подмножество. Здесь сначала мы должны проверить, принадлежат ли два подмножества одному и тому же множеству. Если нет, то мы не можем выполнить объединение.

Идея: Первоначально создайте подмножества, содержащие только один узел, который является родителем самого себя. Теперь при обходе ребер, если два конечных узла ребра принадлежат одному и тому же множеству, они образуют цикл. В противном случае выполните объединение, чтобы объединить подмножества вместе.

Сложность: в худшем случае занимает $O(n)$, но может быть улучшена до $O(\log(n))$.

Оптимизация

Объединение по рангу: Идея состоит в том, чтобы всегда прикреплять дерево меньшей глубины под корень более глубокого дерева.

Path Compression: Идея сжатия пути состоит в том, чтобы сделать найденный корень родителем x , чтобы нам не приходилось снова проходить все промежуточные узлы. Если x является корнем поддерева, то путь (к корню) от всех

узлов под x также сжимается.

Сложность: Временная сложность каждой операции становится даже меньше $O(\log(n))$.

Дополнительно: <https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2>, <https://www.programiz.com/dsa/kruskal-algorithm>, <https://www.geeksforgeeks.org/introduction-to-disjoint-set-data-structure-or-union-find-algorithm>, <https://www.geeksforgeeks.org/union-by-rank-and-path-compression-in-union-find-algorithm>

Реализация (C# пример)

Реализацию самого графа можно найти [здесь](#).

```
/// <summary> Класс подмножество для алгоритма Union-find</summary>
private class Subset<TVertex, TEdge> where TVertex : IComparable where TEdge
{
    public Vertex<TVertex, TEdge> Parent { get; set; }

    public int Rank { get; set; }
}

/// <summary> Метод выполняющий поиск, сжатия пути </summary>
private static Vertex<TVertex, TEdge> Find<TVertex, TEdge>(IReadOnlyDictionary<TVertex, Subset<TVertex, TEdge>> subsets, TVertex vertex)
{
    // Идея: сделать найденный корень родителем
    if (subsets[vertex].Parent != vertex)
        subsets[vertex].Parent = Find(subsets, subsets[vertex].Parent);

    return subsets[vertex].Parent;
}

/// <summary> Метод объединения вершин для Union-find </summary>
private static void Union<TVertex, TEdge>(IReadOnlyDictionary<TVertex, Subset<TVertex, TEdge>> subsets, TVertex vertex1, TVertex vertex2)
{
    var xRoot = Find(subsets, vertex1);
    var yRoot = Find(subsets, vertex2);

    // Идея: подкреплять дерево меньшей глубины под корень более глубокого ,
    if (subsets[xRoot].Rank < subsets[yRoot].Rank)
    {
        subsets[xRoot].Parent = yRoot;
    }
    else if (subsets[xRoot].Rank > subsets[yRoot].Rank)
    {
        subsets[yRoot].Parent = xRoot;
    }
    else
    {
        // Идея: если равны, то можно выбрать любой из корней
        subsets[xRoot].Parent = yRoot;
    }
}
```

```

    {
        subsets[yRoot].Parent = xRoot;
        ++subsets[xRoot].Rank;
    }
}

public static (IList<Edge<int, TVertex>>, int) Kruskal<TVertex>(this Graph<
{
    var verticesCount = graph.Vertices.Count;
    var edges = graph.Edges.ToList();
    var result = new List<Edge<int, TVertex>>();

    var edgesCounter = 0;
    var verticesCounter = 0;
    var minCost = 0;

    edges.Sort();

    var subsets = new System.Collections.Generic.Dictionary<Vertex<TVertex,

foreach (var vertex in graph.Vertices) // Массив множеств, вершины сами
{
    subsets.Add(vertex, new Subset<TVertex, int>() {Parent = vertex, Ra
}

while (verticesCounter < verticesCount - 1)
{
    var nextEdge = edges[edgesCounter++];

    var x = Find(subsets, nextEdge.InitialVertex);
    var y = Find(subsets, nextEdge.DestinationVertex);

    if (x != y)
    {
        result.Add(nextEdge);
        minCost += nextEdge.Weight;
        verticesCounter++;

        Union(subsets, x, y);
    }
}

return (result, minCost);
}

```

Билет 17: Алгоритм Прима (поиска минимального остовного дерева)

Описание

Как и алгоритм Крускала, также является жадным алгоритмом.

Идея состоит в том, чтобы поддерживать два набора вершин. Первый набор содержит вершины, уже включенные в MST, другой набор содержит еще не включенные вершины. На каждом шаге он рассматривает все ребра, соединяющие два множества, и выбирает ребро с минимальным весом из этих ребер. После выбора ребра он перемещает другую конечную точку ребра в набор, содержащий MST.

Итак, на каждом шаге алгоритма Прима находим разрез (из двух наборов, один содержит вершины, уже включенные в MST, а другой содержит остальные вершины), выбирают ребро минимального веса из разреза и включают эту вершину в Набор MST (набор, содержащий уже включенные вершины).

Алгоритм:

- Создайте набор `mstSet`, который отслеживает вершины, уже включенные в MST;
- Присвойте ключевое значение всем вершинам входного графа. Инициализируйте все значения ключей как `INFINITE`. Присвойте значение ключа как 0 для первой вершины, чтобы она выбиралась первой;
- Пока `mstSet` не включает все вершины:
 - Выберите вершину `u`, которой нет в `mstSet` и которая имеет минимальное значение ключа;
 - Включите `u` в `mstSet`;
 - Обновите ключевое значение всех смежных вершин `u`. Чтобы обновить ключевые значения, выполните итерацию по всем соседним вершинам. Для каждой соседней вершины `v`, если вес ребра `uv` меньше, чем предыдущее значение ключа `v`, обновите значение ключа как вес `uv`;


Сложность: $O(V^2)$.

Дополнительно: <https://www.geeksforgeeks.org/prim-minimum-spanning-tree-mst-greedy-algo-5>, <https://www.programiz.com/dsa/prim-algorithm>

Реализация (C# пример)

Реализацию самого графа можно найти [здесь](#).

```
public static (IList<Edge<int, TVertex>>, int) Prima<TVertex>(this Graph<TV
```



```

var notVisitedVertices = graph.Vertices.ToList();
var visitedVertices = new List<Vertex<TVertex, int>>();

var notVisitedEdges = graph.Edges.ToList();
var visitedEdges = new List<Edge<int, TVertex>>();

if (graph.Vertices.Count > 0)
{
    // Выбираем первую вершину для просмотра
    var visitedVertex = notVisitedVertices[0];
    visitedVertices.Add(visitedVertex);
    notVisitedVertices.Remove(visitedVertex);

    while (notVisitedVertices.Count > 0)
    {
        var indexMinEdge = -1;

        for (var i = 0; i < notVisitedEdges.Count; i++) // Обходим не п
        {
            if (visitedVertices.IndexOf(notVisitedEdges[i].InitialVertex) < 0 &&
                visitedVertices.IndexOf(notVisitedEdges[i].DestinationVertex) < 0)
            {
                if (indexMinEdge != -1)
                {
                    if (notVisitedEdges[i].Weight < notVisitedEdges[indexMinEdge].Weight)
                    {
                        indexMinEdge = i;
                    }
                }
            }
        }

        if (visitedVertices.IndexOf(notVisitedEdges[indexMinEdge].InitialVertex) < 0)
        {
            visitedVertices.Add(notVisitedEdges[indexMinEdge].DestinationVertex);
            notVisitedVertices.Remove(notVisitedEdges[indexMinEdge].DestinationVertex);
        }
        else
        {
            visitedVertices.Add(notVisitedEdges[indexMinEdge].InitialVertex);
            notVisitedVertices.Remove(notVisitedEdges[indexMinEdge].InitialVertex);
        }

        visitedEdges.Add(notVisitedEdges[indexMinEdge]);
        minCost += notVisitedEdges[indexMinEdge].Weight;
        notVisitedEdges.RemoveAt(indexMinEdge);
    }
}

```

```
    return (visitedEdges, minCost);  
}
```

Билет 18: Обход графа в глубину (Depth-First Search, DFS)

Описание

Один из основных методов обхода графа, часто используется для проверки связности, поиска цикла и т.д. Общая идея состоит в том, что для каждой не пройденной вершины необходимо найти все не пройденные смежные вершины и повторить поиск для них.


Реализация: Используем стек. Начинаем с того, что помещаем стартовую вершину графа на вершину стека. Далее берём верхний элемент стека и отмечаем его как посещённый. Все смежные вершины, которых не отмечены как посещённые помещаем в верх стека. Продолжаем повторять эти действия до тех пор пока стек не станет пустым.

Сложность: $O(V + E)$, где V — общее количество вершин, E — общее количество рёбер.

Дополнительно: <https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph>, <https://www.programiz.com/dsa/graph-dfs>

Реализация (C# пример)

Реализацию самого графа можно найти [здесь](#).

```
public static IEnumerable<Vertex<TVertex, TEdge>> Dfs<TVertex, TEdge>(this   
{  
    var stack = new Stack<Vertex<TVertex, TEdge>>();  
    var visitedVertices = new List<Vertex<TVertex, TEdge>>();  
    var visitedEdges = new List<Edge<TEdge, TVertex>>();  
  
    stack.Push(startVertex);  
  
    while (stack.Count > 0)  
    {  
        var currentVertex = stack.Pop();  
        if (visitedVertices.Contains(currentVertex)) continue;  
        visitedVertices.Add(currentVertex);  
  
        yield return currentVertex;  
    }  
}
```

```

foreach (var edge in currentVertex.EdgesList)
{
    if (!visitedEdges.Contains(edge))
    {
        visitedEdges.Add(edge);
        var e1 = edge.InitialVertex == currentVertex ? edge.Destination : edge.InitialVertex;
        if (e1 != null)
        {
            stack.Push(e1);
        }
    }
}
}

```

Билет 19: Обход графа в ширину (Breadth-first search, BFS)

Описание

Суть достаточно проста. Обход начинается с посещения определённой вершины (для обхода всего графа часто выбирается произвольная вершина). Затем алгоритм посещает соседей этой вершины. За ними - соседей соседей, и так далее.

Реализация: Используется очередь. В нее будут закладываться вершины после того, как до них будет определено кратчайшее расстояние. То есть очередь будет содержать вершины, которые были «обнаружены» алгоритмом, но не были рассмотрены исходящие ребра из этих вершин. Из очереди последовательно извлекаются вершины, рассматриваются все исходящие из них ребра. Если ребро ведет в не обнаруженную до этого вершину, то есть расстояние до новой вершины не определено, то оно устанавливается равным на единицу больше, чем расстояние до обрабатываемой вершины, а новая вершина добавляется в конец очереди.

Таким образом, если из очереди извлечена вершина с расстоянием d , то в конец очереди будут добавлены вершины с расстоянием $d + 1$, то есть в любой момент исполнения алгоритма очередь состоит из вершин, удаленных на расстояние d , за которыми следуют вершины, удаленные на расстояние $d + 1$.

Сложность: $O(V + E)$, где V — общее количество вершин, E — общее количество рёбер.

Дополнительно: <https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a->

Реализация (C# пример)

Реализацию самого графа можно найти [здесь](#).

```
public static IEnumerable<Vertex<TVertex, TEdge>> Bfs<TVertex, TEdge>(this 
{
    var queue = new Queue<Vertex<TVertex, TEdge>>();
    var visitedVertices = new List<Vertex<TVertex, TEdge>>();
    var visitedEdges = new List<Edge<TEdge, TVertex>>();

    queue.Enqueue(startVertex);

    while (queue.Count > 0)
    {
        var currentVertex = queue.Dequeue();
        if (visitedVertices.Contains(currentVertex)) continue;
        visitedVertices.Add(currentVertex);

        yield return currentVertex;

        foreach (var edge in currentVertex.EdgesList)
        {
            if (!visitedEdges.Contains(edge))
            {
                visitedEdges.Add(edge);
                var el = edge.InitialVertex == currentVertex ? edge.Destination : edge.InitialVertex;
                if (el != null)
                {
                    queue.Enqueue(el);
                }
            }
        }
    }
}
```

Билет 20: Алгоритм Дейкстры (поиск кратчайшего пути) (Dijkstra's algorithm)

Описание

Алгоритм, который находит кратчайший путь от одной вершины графа до другой. Графы используют для моделирования реальных объектов, а алгоритмы поиска пути — при их изучении, а также решении практических задач. Алгоритм Дейкстры

работает для графов, у которых нет ребер с отрицательным весом, т.е. таких, при прохождении через которые длина пути как бы уменьшается. В отличие от похожих алгоритмов, алгоритм Дейкстры ищет оптимальный маршрут от одной заданной вершины ко всем остальным. Попутно он высчитывает длину пути — суммарный вес ребер, по которым проходит при этом маршруте.

Реализация: На каждом шаге алгоритма он раскрывает какую-то вершину в графе. Изначальная цена пути в начальной вершине ноль. На каждом шаге раскрываем вершину у которой оценка минимальна. Когда мы раскрываем вершину, мы просматриваем все смежные с ней и для каждой из них строим свою оценку, прибавлением веса ребра к оценке раскрываемой вершины. Далее смотрим какую вершину раскрывать дальше, тут проявляется жадность алгоритма. Раскрывая выбранную вершину, мы просматриваем все инцидентные ей и обновляем оценку для них. Так продолжаем дальше. Когда мы раскрываем вершину и оказывается, что она конечная алгоритм останавливается. Как собственно найти оптимальный путь? Каждый раз обновляя оценку мы будем запоминать из какой вершины это обновление поступило.

Сложность: $O(V^2)$, где V — количество вершин графа.

Дополнительно: <https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/>, <https://www.programiz.com/dsa/dijkstra-algorithm>

Реализация (C# пример)

Реализацию самого графа можно найти [здесь](#).

```
private class DijkstraData<TVertex, TEdge> where TVertex : IComparable where TEdge : IComparable
{
    public int Price { get; init; }

    public Vertex<TVertex, TEdge>? Previous { get; init; }
}

public static IEnumerable<Vertex<TVertex, int>> Dijkstra<TVertex>(this Graph<TVertex, TEdge> graph, Vertex<TVertex, TEdge> start)
{
    var noVisitedVertices = graph.Vertices.ToList();

    var track = new System.Collections.Generic.Dictionary<Vertex<TVertex, TEdge>, int>();

    track[start] = new DijkstraData<TVertex, int> { Previous = null, Price = 0 };

    while (true)
    {
        Vertex<TVertex, int>? toOpen = null;
        var bestPrice = int.MaxValue;

        foreach (var vertex in noVisitedVertices)
        {
            if (vertex.Previous == null)
            {
                continue;
            }

            var price = track[vertex.Previous].Price + vertex.Edge.Weight;

            if (price < bestPrice)
            {
                bestPrice = price;
                toOpen = vertex;
            }
        }

        if (toOpen == null)
        {
            break;
        }

        track[toOpen] = new DijkstraData<TVertex, int> { Previous = toOpen.Previous, Price = bestPrice };
        noVisitedVertices.Remove(toOpen);
    }

    return track.Values;
}
```

```

foreach (var vertexElement in noVisitedVertices)
{
    if (track.ContainsKey(vertexElement) && track[vertexElement].Pr
    {
        toOpen = vertexElement;
        bestPrice = track[vertexElement].Price;
    }
}

if (toOpen != null && toOpen.Equals(end))
{
    break;
}

if (toOpen != null)
{
    foreach (var edge in toOpen.EdgesList)
    {
        var currentPrice = track[toOpen].Price + edge.Weight;

        var nextVertex = edge.InitialVertex.Equals(toOpen) ? edge.D

        if (!track.ContainsKey(nextVertex) || track[nextVertex].Pri
        {
            track[nextVertex] = new DijkstraData<TVertex, int> {Pri
        }
    }

    noVisitedVertices.Remove(toOpen);
}
}

var result = new List<Vertex<TVertex, int>>();


var tmp = end;
while (tmp != null)
{
    result.Add(tmp);
    tmp = track[tmp].Previous;
}

result.Reverse();

return result;
}

```

Releases 1

 **v1.0.0** Latest
on Jan 17, 2023

Languages

 **C#** 100.0%

[Terms](#) [Privacy](#) [Security](#) [Status](#) [Docs](#) [Contact](#) [Manage cookies](#) [Do not share my personal information](#)



© 2024 GitHub, Inc.